# Travaux pratiques du DEA CPM
# Initiation aux calculs QCD sur réseaux

P.A.M. Guichon
*Irfu/SPhN CEA Saclay F91191 Gif sur Yvette*

January 16, 2009

# Contents

# Préliminaires

## 0.1 Introduction

Le premier TP est une introduction aux méthodes d'intégration stochastiques. Le second une initiation aux calculs QCD sur réseaux par la pratique. Le contrôle sera fait à partir des comptes rendus (voir Section 2.11)

## 0.2 Organisation

Il y aura 2 séances de TP encadrées qui auront lieu Mardi 3/2/09 et 10/2/09 après-midi dans la salle bleue (enseignement informatique) du LAL où les étudiants disposeront du matériel nécessaire (Imacs fonctionnant sous MacOS). Ces séances comporteront une petite introduction aux différents concepts qui seront ensuite mis en pratique. Ces 2 séances sont certainement insuffisantes pour mener à terme un projet de calcul sur réseau non trivial. Au mieux elles serviront à mettre les étudiants sur les bons rails. Pour aboutir à un résultat satisfaisant ils devront fournir un travail personnel en dehors de ces séances.

Ceux qui disposent d'un PC avec linux ou d'un Mac sous OS pourront faire ce travail sur leur propre machine. Il suffira d'installer les codes (venir en TP avec une clef USB pour copier les sources). Ceux qui n'ont que Windows sont fortement encouragés à installer linux en double boot (Ubuntu ou Kubuntu par exemple, gratuits et faciles à installer).

Ceux qui ne disposent d'aucun matériel pourront travailler avec les clusters lx2 et lx4 du LAL s'ils ont rempli la demande de login. Les codes sont déjà installés sur ces clusters et l'accès se fera par internet sécurisé (ssh -Y nom@lx2.lal.in2p3.fr) depuis les PC linux en libre service de l'IPN. On pourra aussi y accèder depuis chez soi à condition d'avoir l'ADSL et un PC capable d'établir une connection ssh et de recevoir le display graphique. Sous linux et Mac-OS , ceci fait partie de l'installation de base. Sous Windows rien n'est garanti.

Pour ceux qui ont besoin d'aide , tous les lundi de 14h à 16h [1]du 23/02 jusqu'au 16/03, la salle bleue sera ouverte et les Imacs disponibles et un encadrement sera assuré.

[1] ou de 14h a 16h, a déterminer pendant la première séance de TP

## 0.3 Connaissances requises

Le codes à développer supposent une minimum de connaissance du C ou du C++. Pour des raisons pratiques le travail se fera en C++ mais, au niveau de programmation utilisé, les différences sont mineures. Le texte du TP fournira plusieurs exemples détaillés.

Le premier TP suppose une certaine familiarité avec la statistique. Il y aura seulement un rappel des notions esssentielles. En règle générale il n'y aura pas de démonstration formelles mais plutôt une présentation avec justification qualitative des résultats qu'on mettra ensuite en pratique. Les étudiants intéressés par les aspects formels pourront consulter les références données dans le texte du TP1.

Le second TP demande surtout d'avoir bien assimilé le cours de QCD de C. Roiesnel et le TP1.

## 0.4 Quelques questions pratiques

Les machines Imacs fonctionnent sous MacOS, lequel est basé sur la version BSD d'UNIX. L'OS Linux fonctionne de façon pratiquement identique, ce qui permet de passer de l'un à l'autre sans trop d'efforts. Evidemment l'environnement graphique est différent mais c'est un détail.

Pour lancer une commande ou une application qui n'apparait pas dans un menu ou dans un icône il suffit d'ouvir un terminal et de taper la commande. Par exemple:

- `prompt>xmgrace &` lance Grace et le détache du terminal.
- `prompt>g++ nom_de_fichier.cpp` compile *nom_de_fichier.cpp* et produit l'executable *a.out* que l'on pourra lancer avec:
- `prompt>./a.out`
- `prompt>tar -cvf archiv.tar directory` archive le contenu du directory dans `archiv.tar`

Pour éviter la pagaille dans les fichiers qui s'accumulent assez vite il faut créer une arborescence à partir du directory HOME de l'utilisateur. Dans une fenêtre terminal faire:

- `echo $HOME` Ceci donne le chemin absolu de l'utilisateur
- `cd` : place dans le répertoire `$HOME`
- `pwd` : normalement le résultat est identique à $HOME, sinon il y a un problème
- `mkdir VOTRENOM` : crée le répertoire VOTRENOM . C'est ce directory qu'il faudra archiver. Une clef USB de 128M est suffisante pour la sauvegarde.

- `cd VOTRENOM`
- `mkdir TP1`
- `cd TP1`
- `mkdir EX01`
- `etc...`
- `mkdir EX05`
- `cd ..` : remonte d'un cran dans l'arborescence
- `mkdir TP2` c'est là qu'on utilisera FermiQCD
- `cd $HOME`

## 0.5 Les instructions du pré-processeur (include)

Quand on lance la compilation avec gcc ou g++, le compilateur effectue d'abord un pré-traitement du programme qui consiste à remplacer les lignes de commandes du type: `#include <system_code>` ou `#include "personnal_code"` par le code source correspondant. En particulier pour disposer des fonctions d'entrée/sortie, des fonctions mathématiques etc... le code doit commencer par cette série d'include:

```
#include <iostream>
#include <fstream>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include <string>
using namespace std;
using std::cout;
#define endl "\n"
```

qu'on peut sauver dans un fichier qu'on nommera par exemple MyHeaders.h. Il suffira alors d'ajouter la ligne:

```
#include "MyHeaders.h"
```

au début de chaque programme. Il faut indiquer au compilateur à quel endroit il peut trouver ce fichier:

```
g++ nom_de_fichier.cpp -I chemin
```

Si le fichier `MyHeaders.h` est dans le répertoire où la commande de compilation est lancée, il suffit de taper

```
g++ nom_de_fichier.cpp -I .
```

car le point est le raccourci pour le répertoire courant. Pour simplifier on copiera ce fichier dans tous les répertoires où on en a besoin (EXO1, EXO2, ...)

Dans le directory *dea*=/home/sluser/TP2/TP_Imac/UseFermiQCD/DEA se trouvent les codes développés pour ce TP. Normalement ce sont les seuls codes que les étudiants doivent éventuellement modifier.

*Ne pas toucher aux fichiers qui sont au dessus ce ce directory, sauf à vos risques et périls.*

Dans *dea* on trouve les directory `Thermalisation, Autocorrelation et Confinement`. Ils contiennent chacun un code source `nom.cpp` et un MakeFile. Pour commencer faire

- `make clean`
- `make`
- `./a.out`

La compilation dure quelques minutes.

# Chapter 1

# TP1: Initiation aux méthodes d'intégration stochastiques

## 1.1 Introduction

Comme on l'a vu dans le cours, le calcul des observables dans QCD sur réseau se ramène à calculer des intégrales du genre:

$$\frac{1}{Z}\int [dU] e^{-S(U)} f(U) \text{ avec } Z = \int [dU] e^{-S(U)} \tag{1.1}$$

où $S(U)$ est l'action euclidienne de QCD, c'est à dire un nombre réel. La notation symbolique $\int [dU]$ représente une intégration sur les matrices $U(i)$ associées à chaque lien $i$ du réseau, c'est à dire:

$$\int [dU] = \int dU(1) dU(2) ... dU(N_l) \tag{1.2}$$

où $N_l$ est le nombre de lien du réseau. Pour un réseau à $D$ dimensions dont le nombre de sites dans les directions $[(1, 2, ...D]$ est $[s_1, s_2, ...s_D]$, on a:

$$N_l = \sum_{i=1,D} (s_i - 1) \Pi_{i \neq j} s_j \tag{1.3}$$

Si $s_1 = s_2 = ... = s_D = s \gg 1$ on a l'expression approchée $N_l \simeq D s^D$. Dans le cas à 4 dimensions et pour $s = 10$, ce qui est une valeur plutôt petite, on obtient $N_l \simeq 40\,000$. D'autre part l'intégration sur chaque $U(i)$ se fait dans le groupe de jauge $SU(N)$ dont le nombre de paramètres réels est $N^2 - 1$. Dans le cas de $SU(2)$ qui sera considéré dans le second TP[1], on voit qu'on se retrouve avec une intégration à 120 000 variables, et c'est plutôt un minimum. Les méthodes d'intégration classiques de type Simpson ou Gauss-Legendre, dont le temps de

[1] La bibliothèque FermiQCD qui sera utilisée dans le second TP permet de faire le calcul pour $N$ quelconque. Le choix $N = 2$ permet d'avoir un temps de calcul raisonnable.

calcul croît exponentiellement avec le nombre de variables, sont donc inutilisables. Les seules méthodes opérationnelles sont les méthodes stochastiques car elles fournissent, pour $N$ tirages, un résultat approché dont l'erreur décroit comme $1/\sqrt{N}$, indépendamment du nombre de variables.

L'objet de ce TP est de s'initier à certaines méthodes stochastiques utilisées dans les calculs sur réseau. Nous le ferons avec des intégrales à 1, 2 ou 3 dimensions car, d'une part, c'est suffisant pour introduire les concepts importants (échantillonage, chaîne de Markov, autocorrélation, erreur Jackknife), et d'autre part, le faible nombre de dimensions permet de faire les calculs en quelques secondes.

## 1.2 Intégration Monte-Carlo

Dans cette introduction nous allons donc considèrer des intégrales du genre

$$I = \frac{1}{Z}\int_{\mathcal{D}} dx e^{-S(x)} f(x), \;\; Z = \int_{\mathcal{D}} dx e^{-S(x)} \tag{1.4}$$

où la variable $x$ peut désigner une ou plusieurs variables réelles. On notera que $I$ représente la valeur moyenne de la fonction $f$(x) pondérée par la probabilité (normalisée) $\exp[-S(x)]/Z$. Dans les calculs sur réseaux c'est toujours ce genre d'intégrale qu'on rencontre.

La méthode de Monte-Carlo la plus simple consiste à faire $N$ tirages aléatoires uniformes de la variable $x$ dans le domaine d'intégration $D$. Soit $\{x_1, x_2, ...x_N\}$ les résultats du tirage. Alors pour une fonction $F(x)$ quelconque, on a:

$$\int dx\, F(x) = \frac{1}{N}\sum_{i=1,N} F(x_i) + O\left(1/\sqrt{N}\right). \tag{1.5}$$

Dans le cas d' un grand nombre de dimension cette approche n'est pas utilisable pour estimer l'intégrale (1.4). En effet le tirage uniforme ignore le fait que l'intégrale est dominée par les régions de $x$ où $\exp[-S(x)]$ est grand, de sorte que la majorité des tirages ne servent à rien et la convergence est très lente. Pour éviter cela on remplace le tirage uniforme par un tirage dont la distribution "suit" la fonction $\exp[-S(x)]$. C'est la méthode d'échantillonage (importance sampling) qui est expliquée dans la Section 1.3.

### Exercice 1:

Bien que cette méthode ne soit pas utilisée dans la suite, c'est l'occasion vérifier que le système d'exploitation fonctionne et de tester le générateur aléatoire. On demande de calculer les intégrales

$$I_n = \int_0^1 dx e^{-x^2} x^n, \;\; n = 0, 2$$

par la méthode MC. Les valeurs exactes sont $I_0 = 0.746824$, $I_2 = 0.189472$.

Normalement l'OS fournit une grande variété de générateurs. Pour cet exercice et ceux qui suivent on se contentera de `random()` qui fournit un entier au hazard entre 0 et $2^{31}$. On commence par faire une fonction qui tire au hazard entre a et b et on la teste avec la valeur moyenne. On enregistre 100 000 tirages dans un fichier `random.dat` dont on fera l' histogramme avec le logiciel Grace pour vérifier que la distribution est bien uniforme. Pour lire les données dans Grace , cliquer sur les menus: `Data->Import->ASCII` et pour l'histogramme: `Data->Transformations->Histogram`)

Le début du programme peut donc ressembler à ce qui suit:

```
// PROGRAM1:  Use and test of random generator
//
#include "MyHeaders.h"            //voir Préliminaires
ofstream outfile("random.dat");   //open stream for output
//
float myrandom(float a,float b)      //random number in [a,b]
     {
      float range=pow(2.,31);
      return a+(b-a)*random()/range;
     }
//
int main()
{
float x,sum=0.,mean;
int i,hitnb=100000;                //make hitnb calls
cout<<"create "<< hitnb<<" rand.  numb.  in [0,1]\
                  and compute mean"<<endl;
for (i=1;i<=hitnb;i++)
   {
    x=myrandom(0.,1.);
    outfile<<i<<" "<<x<<endl;     //write file for histogram
    sum=sum+x;
    mean=sum/i;
    if(i%(hitnb/100)==0)
         {
          cout<<"i= "<<i<<" mean= "<<mean<<endl;
         }
   }
outfile.close();
//                   calcul des intégrales...
}                       //end of PROGRAM1
```

## 1.3 Echantillonage représentatif (importance sampling)

Supposons qu'on sache créer une suite $\{x_1, x_2, ...x_N\}$ dont la distribution de probabilité est $\exp[-S(x)]$. Celà veut dire que le nombre de points situés entre $x$ et $x + dx$ est proportionnel à $\exp[-S(x)]$, ce qui peut être testé en faisant l'histogramme. Alors on peut écrire, pour $N \to \infty$:

$$\overline{f} = \frac{1}{N}\sum_i f(x_i) \to \frac{1}{Z}\int_{\mathcal{D}} dx e^{-S(x)} f(x) \tag{1.6}$$

et, si les points $x_i$ sont statistiquement indépendants, l'erreur (dite erreur standard) est:

$$\sigma = \frac{1}{N}\sqrt{\sum_i (f(x_i) - \bar{f})^2} = \sqrt{\frac{\overline{f^2} - \overline{f}^2}{N}} \tag{1.7}$$

On verra plus loin que si les points $x_i$ sont corrélés[2], la vraie erreur peut être beaucoup plus grande que $\sigma$.

### 1.3.1 Chaîne de Markov

Une chaîne chaîne de Markov $\{X_1, X_2, ...\}$ est une suite de variables aléatoires telle que la distribution de $X_{n+1}$ ne dépend que de celle de la précédente $X_n$. Pour 2 variables aléatoires $(X, Y)$ de la chaîne on note $P(x \to y)$ la probabilité d'avoir $Y = y$ sachant que $X = x$. Si $X$ est distribuée suivant $M(x)$ alors la distribution de $Y$ est:

$$\int dx M(x) P(x \to y).$$

Par hypothèse la probabilité est normalisée suivant:

$$\int P(x \to y) dy = 1$$

Si on impose de plus que $P$ soit strictement positive et vérifie la condition de micro-réversibilité (ou balance détaillée):

$$e^{-S(x)} P(x \to y) = e^{-S(y)} P(y \to x) \tag{1.8}$$

alors on peut montrer que la distribution des variables de la chaîne converge vers $\exp[-S(x)]$. On trouvera la démonstration dans la reférence [1].

Développons un peu la signification de ce résultat. La chaîne de Markov est une suite de variables aléatoires $\{X_1, X_2, ...\}$ qui acquièrent des valeurs $\{x_1, x_2, ...\}$ lors d'un processus aléatoire (défini en pratique par un algorithm).

[2] Dans la méthode de Monte Carlo simple les $x_i$ sont statistiquement indépendant, si le tirage est bien uniforme. Cela ne dépend que du générateur de nombres pseudo-aléatoires.

Pour chaque réalisation du processus on obtient une chaîne différente. Supposons qu'on réalise $M$ fois le processus, avec $M$ très grand. On va obtenir:

$$\begin{array}{lccccc} & \{X_1 & X_2 & ... & X_n & ...\} \\ \text{Processus 1} & \{x_1^{(1)} & x_2^{(1)} & ... & x_n^{(1)} & ...\} \\ \text{Processus 2} & \{x_1^{(2)} & x_2^{(2)} & ... & x_n^{(2)} & ...\} \\ ... & & & ... & & \\ \text{Processus } M & \{x_1^{(M)} & x_2^{(M)} & ... & x_n^{(M)} & ...\} \end{array}$$

Si on considère les valeurs[3] $\{x_n^{(1)}, x_n^{(2)}, ..., x_n^{(M)}\}$ prises par n'importe quelle variable $X_n$, le théorème de convergence dit qu' elles sont toutes distribuées suivant $\exp[-S(x)]$ si $n$ est suffisamment grand. On dit que la chaîne est thermalisée à partir de $n$.

Evidemment on ne veut pas construire un infinité de chaînes de Markov et ce n'est pas nécessaire. En effet considérons les valeurs $\{x_1^{(1)}, x_2^{(1)}, ...x_n^{(1)}, x_{n+1}^{(1)}...\}$ obtenues par le premier processus. Puiqu'à partir de $n$ on a la même distribution $\exp[-S(x)]$, les points $\{x_n^{(1)}, x_{n+1}^{(1)}...\}$sont forçément distribués suivant cette loi. Il suffit donc de construire une seule chaîne de Markov $\{x_1, x_2, ...x_n, ...\}$ pour obtenir l'échantillonnage désiré et on pourra écrire

$$\frac{1}{Z}\int_{\mathcal{D}} dx e^{-S(x)} f(x) \simeq \frac{1}{N}\left[f(x_n) + f(x_{n+1}) + ...f(x_{n+N})\right]. \tag{1.9}$$

La valeur de thermalisation $n$ s'obtient, par exemple, en augmentant sa valeur jusqu'à obtenir la stabilisation de la somme qui apparait dans l'éq.(1.9). Dans les exemples que nous allons considèrer, $N$ est suffisamment grand pour que la contribution des points non thermalisés

$$\frac{1}{N}\left[f(x_1) + f(x_2) + ...f(x_{n-1})\right]$$

soit négligeable. On pourra donc ignorer le problème de la thermalisation et garder toute la chaîne dans le calcul de la valeur moyenne.

Les différents algorithms se distinguent par la facon de tirer la valeur $x_{k+1}$ de $X_{k+1}$ connaissant valeur $x_k$ de la variable précédente. Les plus utilisés sont les algorithms du bain thermique et de Metropolis.

## 1.4 Bain thermique

### 1.4.1 Algorithm

Pour fixer les idées supposons que $x$ représente l'ensemble des matrices de lien du réseau $\{U_1, U_2, ...\}$. Le bain thermique consiste à fixer toutes les matrices de lien sauf une, disons $U_1$ , et de générer la nouvelle valeur de celle ci avec une probabilité proportionelle à $\exp[-S(U_1, U_2, ...)]$. On repète ensuite cette étape pour tous les liens. L'algorithm de mise à jour est le suivant:

[3] c'est à dire la colonne d'indice n dans le tableau

1. On tire une valeur $U_1$ au hazard dans le domaine d'intégration (ici le groupe de jauge).
2. On tire au hazard un nombre $\eta$ entre 0 et 1 (plus généralement entre la borne inférieure et la borne supérieure de la distribution qu'on veut générer.)
3. Si $\eta > \exp[-S(U_1, U_2, ...)]$ le tirage est rejeté et on recommence à l'étape 1, sinon on accepte la valeur et on passe au lien suivant.

Du fait que $U_1$ est tirée au hazard sans référence à sa valeur précédente dans la chaîne de Markov, le tirage est presque toujours rejeté car la fonction $\exp[-S(U_1, U_2, ...)]$ est très piquée autour de certaine valeurs de $U_1$ et presque nulle partout ailleurs. C'est pourquoi cette méthode n'est pas utilisable de façon générale. Dans le cas de QCD exploite la structure de l'action $S$ pour faire des changements de variables qui permettent d'étaler les pics sur tout le domaine d'intégration. C'est cette méthode qui est mise en oeuvre dans la bibliothèque FermiQCD. Le nom "bain thermique" vient du fait que les liens $\{U_2, U_3, ...\}$ jouent le rôle de thermostat avec lequel on met le lien $U_1$ en équilibre thermique.

### 1.4.2 Exercice

A titre d'illustration on propose de vérifier le fonctionnement de l'algorithm pour une variable à 1 dimension. Dans ce cas le problème de la convergence est anodin. On demande de calculer le rapport $I_2/I_0$. (Pourquoi ne peut on pas calculer $I_2$ et $I_0$?). Le programme doit:

1. Générer une chaîne de Markov suivant l'algorithm ci dessus.
2. Enregistrer chaque nouvelle valeur acceptée pour faire l'histogramme.
3. Calculer la nouvelle valeur du rapport $I_2/I_0$ à chaque nouveau point ajouté.
4. Calculer l'erreur standard.
5. Ecrire dans un fichier la longeur de la chaîne, l'intégrale et l'erreur pour en faire un graphique (voir figure 1.1).

Le programme est le suivant:

```
// PROGRAM2 use of heat bath algorithm for a 1d integral
//
#include "MyHeaders.h"
//
ofstream outfile1("calcul.dat");
ofstream outfile2("sample.dat");
//
// myrandom:  random number in [a,b]
//
// prob:  unnormalized probability distribution
    float prob(float x){return exp(-x*x);}
//
// func:  function multiplying the probability
    float func(float x){return x*x;}
//
int main(){
int i,nmarkov=100000;
float x_min=0.,x_max=1.,eta;
float x,sum=0.,sum2=0.,mean,mean2,stderror;
//
for( i=1;i<=nmarkov;i++)
   {
    do
       {
        x=myrandom(x_min,x_max);
        eta= myrandom(0.,1.);
       }while(eta>prob(x));    // get x distributed as prob(x)
//
   sum=sum+func(x);
   sum2=sum2+func(x)*func(x);
   mean=sum/i;
   mean2=sum2/i;
   stderror=sqrt((mean2-mean*mean)/i);
   if(i%1000==0)
      {
       cout<<i<<" I2/I0:  "<<mean<<" std error:  "\
           <<stderror<<" exact:  0.2537"<<endl;
       outfile1<<i<<" "<<mean<<" "<<stderror<<endl;
      }
   outfile2<<i<<" "<<x<<endl;
   }
outfile1.close(),outfile2.close();
}
```

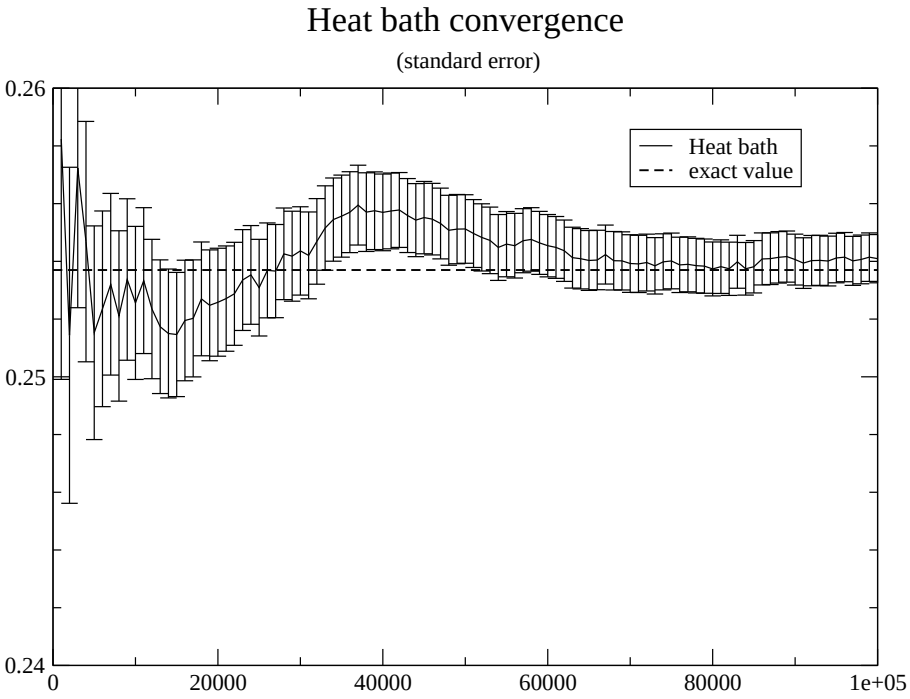


Figure 1.1: Convergence

## 1.5 Metropolis

### 1.5.1 Algorithm

L'algorithm, dans sa version la plus simple, est le suivant:

1. On choisit une première valeur $x_1$ pour la variable $X_1$.
2. On suppose connue la valeur $x_k$ de la variable $X_k$. On tire de facon uniforme la variable $X_{k+1}$dans un domaine $\Delta$, qui dépend à priori de $x_k$ et qui est spécifié plus bas. Soit $x_{k+1}$ le résultat. On pose
$$R = \frac{\exp(-S(x_{n+1}))}{\exp(-S(x_n))}. \tag{1.10}$$
3. Si $R > 1$, on attribue à $X_{k+1}$la valeur $x_{k+1}$.
4. Si $R < 1$, on attribue à $X_{k+1}$ la valeur $x_{k+1}$ avec la probabilité $R$ et la valeur $x_k$ avec la probabilite $1 - R$. En pratique on tire un nombre $\eta$ au hazard (distribution uniforme) dans $[0, 1]$ et on retient $x_{k+1}$ si $R > \eta$ et $x_k$ dans le cas contraire.
5. On recommence à l'étape 2 avec $k \to k + 1$

Le domaine de tirage $\Delta$ dépend du problème considèré. Le choix le plus courant, est de faire le tirage de $X_{k+1}$ "autour" de la valeur précédente $x_k$. Si $x$ est une variable (ou un ensemble de variables) réelle on tire $X_{k+1}$ de facon uniforme dans $\Delta = [x_k - \epsilon, x_k + \epsilon]$, avec $\epsilon$ ajusté pour que le taux d'acceptation soit d'environ 50%. La méthode est plutôt adaptée à un domaine d'intégration sans frontière (pas forcement infini) sinon la nouvelle valeur risque d'être en dehors du domaine, ce qui génère des effets de bords.

***Remarque:*** *Si $x$ représente une matrice de lien, on ne peut pas lui ajouter quelque chose car le résultat n'est pas dans le groupe. La méthode utilisée dans ce cas est la multiplier par une matrice tirée au hazard dans un sous ensemble suffisamment représentatif du groupe.*

### 1.5.2 Exercice 3

Pour corser un peu la chose on considère une intégration à $D$ dimensions. Pour ne pas avoir d'ennuis avec les effets de bord on va calculer le rapport:

$$r = \frac{\int_{-\infty}^{\infty} dx x^2 e^{-x^2}}{\int_{-\infty}^{\infty} dx e^{-x^2}} = \frac{D}{2}$$

où cette fois $x$ désigne un vecteur à $D$ dimensions, c'est à dire:

$$dx = dx_1 dx_2 ... dx_D, \;\; x^2 = \sum_{i=1,D} x_i^2$$

On adapte le programme précédent pour faire ce calcul en faisant des fonctions `prob et func` qui dépendent de la dimension. Il faut ausi ajouter un compteur pour enregistrer le nombre d'acceptations ( c'est à dire les cas où $R > \eta$) et la variable variable $\epsilon$ qu'on ajuste pour que le taux d'acceptation soit de l'ordre de 50%.

On commencera avec le cas $D = 1$. Essayer 10 000 puis 100 000 tirages. Tracer l'évolution de la valeur calculée en fonction de la longueur de la chaîne. Observer que l'erreur standard peut sous estimer l'écart par rapport à la valeur exacte. Au passage on enregistre les valeurs successives de $x$ (`sample.dat`) pour faire l'histogramme et vérifier que la distribution suit bien la probabilité $\exp(-x^2)$. Les figures 1.2 et 1.3 montrent ce qu'on peut obtenir.

En 2 dimensions on peut s'amuser à visualiser la chaîne (`walk_2d.dat`). Prendre seulement 10 000 points sinon le graphe risque d'être tout noir! On peut aussi choisir une probabilité avec 2 maxima et voir que la chaine peut être piègée sur un des maxima suivant la valeur choisie pour le paramètre $\epsilon$. Etudier ensuite comment le calcul converge en fonction de la dimension. Voilà le programme:

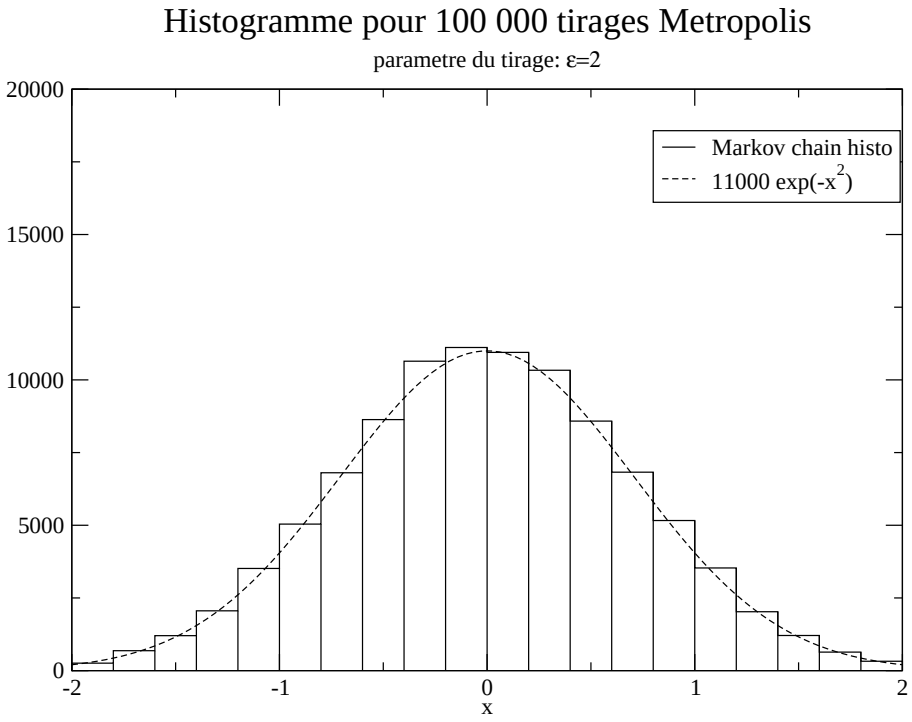


Figure 1.2: Histogramme

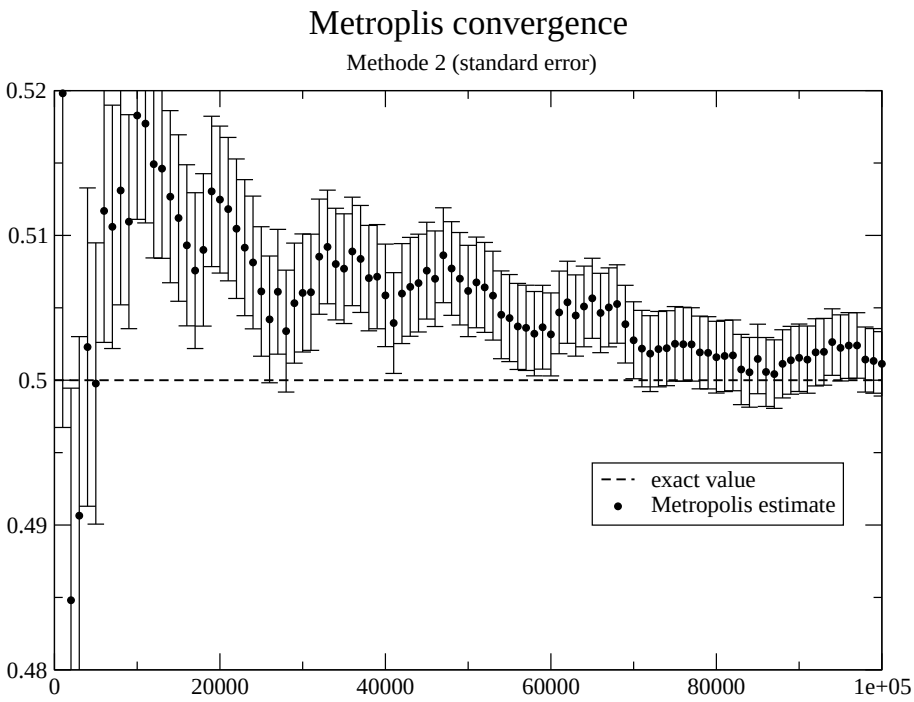


Figure 1.3: Convergence

```
// Program3:  Integration in D dimensions without bounds
//////////////////////////////////////////////////////
#include "MyHeaders.h"
//
ofstream outfile1("calcul_Nd.dat");
ofstream outfile2("sample.dat");
ofstream outfile3("walk_2d.dat");
//FUNCTIONS:
// myrandom:  random number in [a,b]//
//////////////////////////////////////////////////
// prob:  unnormalized probability distribution in D/
//////////////////////////////////////////////////
     float prob(int d,float *x){
     float arg=0.;
     for(int i=0;i<d;i++){arg=arg+x[i]*x[i];}
     return exp(-arg);}
/////////////////////////////////////////
// func:  function multiplying the proba//
/////////////////////////////////////////
     float func(int d,float *x){
     float arg=0.;
     for(int i=0;i<d;i++){arg=arg+x[i]*x[i];}
     return arg;}
////////
//MAIN//
////////
int main(){
int j,dim=2; // set dimension
float epsi=1.; // set parameter for update
//////////////////////////////////////
//2D MARKOV CHAIN. Store function//
//////////////////////////////////////
int i,nmarkov=100000;
float x[dim],y[dim];
float store_func[nmarkov+1];
float accept=0.;
for(j=0;j<dim;j++)
   {
    x[j]=0.;
   } // first point of markov chain
//
for( i=1;i<=nmarkov;i++)
   {
   for(j=0;j<dim;j++)
      {
       y[j]=x[j]+myrandom(-epsi,epsi);
```

```
      }
   if( prob(dim,y)/prob(dim,x) > myrandom(0.,1.)  )
      {
       for(j=0;j<dim;j++) x[j]=y[j];accept=accept+1.;
      }
   store_func[i]=func(dim,x);
   if(dim==1)            // write 1d chain for histogram
      {
       outfile2<<i<<" "<<x[0]<<endl;
      }
   if(dim==2)           // write 2d chain to watch walk
      {
       outfile3<<x[0]<<" "<<x[1]<<endl;
      }
   }
////////////////////////////
//MEAN AND STANDARD ERROR//
////////////////////////////
float sum=0.,mean,sum2=0.,mean2,sigma,exact=dim*0.5;
for(i=1;i<=nmarkov;i++)
   {
    sum=sum+store_func[i];mean=sum/i;
    sum2=sum2+store_func[i]*store_func[i];mean2=sum2/i;
    sigma=sqrt((mean2-mean*mean)/i);
    if(i%1000==0)
       {
        outfile1<<i<<" "\          //write result
        <<mean<<" "<<sigma<<endl;
       }
   }
//
cout<<" calcul en dimension:  "<<dim<<endl;
cout<<" accept rate:  "<<accept/nmarkov<<endl;
cout<<"mean:  "<<mean<<" std error:  "\
     <<sigma<<" exact:  "<<exact<<endl;
//
outfile1.close();outfile2.close();outfile3.close();
}                                        //End of Program3
```

## 1.6 Autocorrélation

### 1.6.1 Introduction

Comme on commence à le voir dans l'exercice précédent, l'erreur standard peut sous-estimer la véritable erreur. Celà est du au fait que les points successifs de l'échantillon ne sont pas statistiquement indépendants. Dans la chaîne de Markov, un point est construit à partir du précédent et donc il existe une certaine corrélation entre eux. Le problème est d'estimer combien d'itérations sont nécessaires pour que cette corrélation disparaisse. Comme le temps de calcul est proportionnel au nombre d'itérations, on parle de temps d'autocorrélation, qu'il ne faut évidemment pas confondre avec le temps physique.

Pour une chaîne $\{x_1, x_2, ...x_N\}$ de longueur $N$ (grand) on définit la fonction d'autocorrélation:

$$\chi(\tau) = \frac{1}{N-\tau} \sum_{i=1,N-\tau} f(x_i) f(x_{i+\tau}) - \bar{f}^2 \tag{1.11}$$

où $f(x)$ est la fonction qui apparait dans l'expression

$$I = \frac{1}{Z} \int_{\mathcal{D}} dx e^{-S(x)} f(x) \simeq \frac{1}{N} \sum_i f(x_i) = \bar{f}$$

Par exemple dans l'exercice précédent, $f(x) = x^2$. En comparant à (1.7) on voit que $\chi(0) = N\sigma^2$ et on introduit la fonction d'autocorrélation normalisée:

$$A(\tau) = \frac{\chi(\tau)}{\chi(0)}. \tag{1.12}$$

On peut montrer que $A(\tau) \sim \exp(-\tau/\tau_e)$ quand $\tau \to \infty$, où $\tau_e$ est le temps caractéristique de la décroissance appelé *temps exponentiel.* Comme pour $\tau$ grand on s'attend à ce que les valeurs $f(x_i)$ et $f(x_{i+\tau})$ soient sans corrélation, la décroissance de $A(\tau)$ indique combien d'itérations sont nécessaires pour supprimer les corrélations. C'est ce qu'on propose d'étudier dans l'exercice suivant.

Auparavant il faut comprendre dans quel sens l'autocorrélation est une source d'erreur. En effet, si la chaîne de Markov est suffisammnent longue, on est assuré que les points (thermalisés) $x_i$ de la chaîne sont distribués suivant la probabilité $\exp(-S)$ et donc on a bien le droit d'écrire:

$$I \to \frac{1}{N} \sum_i f(x_i) \ \text{ qd } N \to \infty.$$

Mais, en pratique, $N$ est fini et il faut avoir une estimation de l'erreur pour choisir la valeur de $N$ qui donnera la précision souhaitée. Si on a déterminé, à partir de la fonction d'autocorrélation, que 2 points séparés par un nombre d'itérations $\tau = \tau_{decorrel}$ sont approximativement décorrélés, alors on peut construire un échantillon de points statistiquenent indépendants en éliminant les points entre $i$ et $i + \tau_{decorrel}$. On peut alors utiliser l'erreur standard (Eq. 1.7)

et arrêter le calcul quand on a atteint la précision souhaitée. Le prix à payer est que l'on n'utilise qu'une fraction $N/\tau_{decorrel}$ des points de la chaîne et comme l'erreur standard décroit comme la racine carrée du nombre de points, l'erreur va être $\sqrt{\tau_{decorrel}}$ fois plus grande que ce qu'on aurait pu espérer. C'est en ce sens que l' autocorrélation est une source d'erreur. Un bon algorithm est donc celui qui donne un temps de décorrélation proche de 1. Ce n'est généralement pas le cas et il faut s'accomoder de cette augmentation de l'erreur, l'essentiel étant de savoir l'estimer correctement.

Au passage notons un aspect pratique important. Il peut arriver que la génération de la chaîne de Markov utilise beaucoup moins de temps CPU que le calcul de la fonction $f$ . Dans ce cas, si on connait vaguement $\tau_{decorrel}$ , on a intérêt à faire $\tau_{decorrel}$ itérations à vide, c'est à dire sans calculer la fonction. De cette facon on crée un échantillon sans corrélations à moindre prix. Dans les exemples de ce TP le temps de calcul est négligeable et donc le problème ne se pose pas. Par contre il est crucial dans les calculs sur réseaux.

L'observation de la décroissance de la fonction d'autocorrélation permet d'estimer $\tau_{decorrel}$ mais il serait commode d'avoir une estimation de l'erreur qui soit valable même si on utilise des points qui sont corrélés. On peut montrer que cette erreur est

$$\sigma_{vraie} = \sigma\sqrt{2\tau_{int}} \tag{1.13}$$

où $\sigma$ est l'erreur standard et $\tau_{int}$ est le "temps d'autocorrélation intégré" dont l'expression est:

$$\tau_{int} = \frac{1}{2} + \sum_{\tau=1,N-1} \left(1 - \frac{\tau}{N}\right) A(\tau) \tag{1.14}$$

$$\simeq \frac{1}{2} + \sum_{\tau=1,N-1} A(\tau) \tag{1.15}$$

L'expression (1.15) est celle qui est utilisée en pratique car, comme $A(\tau)$ décroit avec $\tau$ le terme en $\tau/N$ est négligeable si $N$ est grand. La valeur de $\tau_{int}$ est comparable au temps $\tau_e$ qui contrôle la décroissance de $A(\tau)$. Lorsque $\tau$ s'approche de $N$, $A(\tau)$ subit de grande fluctuations statistiques car la somme (1.11) contient peu de termes. Il faut donc éventuellement éliminer ces contributions parasites en tronquant la somme 1.14 avant les fluctuations. Le problème se pose seulement si $N$ n'est pas assez grand.

### 1.6.2 Exercice 4

C'est la continuation de l'exercice 3. On rajoute un petit bout qui calcule la fonction d'autocorrélation. Faire tracer cette fonction et éliminer les points qui sont dans les fluctuations (`data->data set operations->drop`) et utiliser la fonction d'integration (`Data->Transformations->Integration`) pour calculer le temps d'autocorrélation intégré. Par combien faut il multiplier l'erreur standard? Essayer plusieurs dimensions. La figure 1.4 montre quelques résultats. Voilà le morceau à ajouter à Program3 et à sauver comme Program4 dans EXO4):

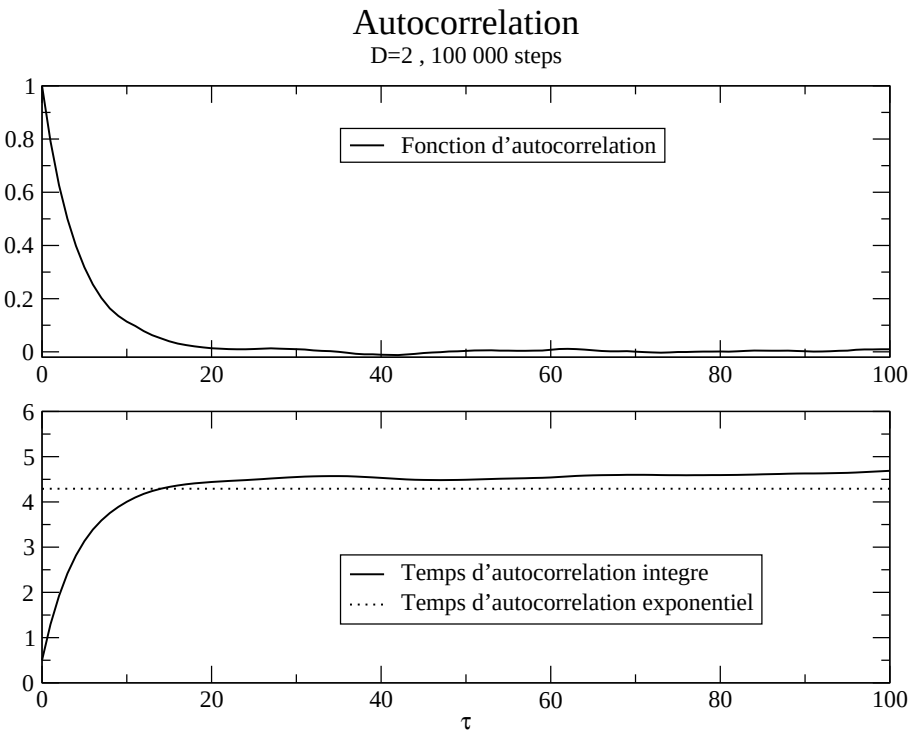
Autocorrelation
D=2 , 100 000 steps
Fonction d'autocorrelation
Temps d'autocorrelation integre
Temps d'autocorrelation exponentiel
τ

Figure 1.4:

```
//////////////////
//AUTOCORRELATION//
//////////////////
ofstream outfile("autocorel_Nd.dat");
float sumshift,chi,chi0,normedchi,rough_time;
int icorel,ncorel=100;
for(icorel=0;icorel<=ncorel;icorel++)
   {
    sumshift=0.;
    for(i=1;i<=nmarkov-icorel;i++)
       {
        sumshift=sumshift+store_func[i]*store_func[i+icorel];
       }
    chi=sumshift/(nmarkov-icorel)-mean*mean;
    if(icorel==0) chi0=chi;
    normedchi=chi/chi0;
    if(icorel==1)
        {
         rough_time=-1/log(normedchi);
        }    // rough estimate of correlation time
    outfile<<icorel<<" "\
           <<normedchi<<endl;       // write correl.  function
   }
cout<<" rough estimate of exponential time:  "\
                        <<rough_time<<endl;
//
outfile.close();
}
```

## 1.7 Erreur Jack knife (couteau de poche)

### 1.7.1 Introduction

L'étude de la fonction d'autocorrélation est assez lourde à mettre en oeuvre. De plus elle doit, en principe, être effectuée chaque fois que l'on change la fonction à intégrer. C'est pourquoi on utilise des méthodes approchées pour estimer la vraie erreur. Il y en a plusieurs mais ici on va se contenter de l'analyse du Jack knife, qui est très populaire.

Le principe est basé sur le binning: supposons par exemple qu'on ait construit une chaîne de $N = 100$ points $\{x_1, x_2, ...x_{100}\}$.Au lieu de calculer la moyenne à partir de $\{f(x_1), f(x_2), ...f(x_{100})\}$ on peut grouper les points par blocs de 10 en définissant les moyennes de chaque bloc:

$$\begin{aligned} \bar{f}_1 &= [f(x_1) + f(x_2) + ... + f(x_{10})]/10, \\ &... \\ \bar{f}_{10} &= [f(x_{91}) + f(x_{92}) + ... + f(x_{100})]/10 \end{aligned}$$

et ensuite calculer la moyenne de $\{\bar{f}_1, ..., \bar{f}_{10}\}$, ce qui donnera évidemment le même résultat que le calcul direct. Mais comme les points des blocs $i$ et $i+1$ sont en moyenne éloignés de 10, les valeurs $\bar{f}_i$ et $\bar{f}_{i+1}$ sont moins corrélées que les valeurs originales. Si le temps d' autocorrélation est plus petit que 10, l'erreur sera pratiquement égale à l'erreur standard, mais calculée avec 10 points au lieu de 100. En pratique on augmente progressivement la taille des blocs (ce qui suppose que le nombre total d'échantillons est assez grand) jusqu'à obtenir un résultat stable. Cette méthode est très fiable mais l'inconvénient est que les blocs peuvent avoir un petit nombre de points, ce qui introduit un biais dans le calcul de l'erreur.

Dans l'analyse du Jack knife la procédure est un peu différente. Au lieu de diviser l'échantillon de 100 en 10 blocs de 10, on construit 10 échantillons de 90 points en ôtant de$\{f(x_1), f(x_2), ...f(x_{100})\}$ un bloc de 10. Le calcul de l'erreur doit évidemment tenir compte du fait que l'on multiplie artificiellement la statistique par 10 mais l'avantage est que les échantillons ainsi construits ont à peu près le même nombre de points que l'original.

Considérons maintenant le cas général: on a un échantillon initial de $N$ valeurs et le divise en $N_b$ blocs de taille $b = N/N_b$. Si $N$ n'est pas un multiple de $N_b$, le dernier bloc sera plus petit que $b$ mais ça ne change rien au résultat. Pour chaque bloc $i$ on définit la moyenne (pour le dernier bloc remplacer $b$ par sa vraie taille)

$$\bar{f}_i = \frac{1}{b} \sum_{k \in bloc\, i} f(x_k)$$

et la moyenne de l'échantillon obtenu en ôtant ce bloc[4] :

$$\hat{f}_i = \frac{1}{N-b} \sum_{k \notin bloc\, i} f(x_k) = \frac{N\bar{f} - b\bar{f}_i}{N-b}.$$

[4]y compris pour le dernier

On a:

$$\bar{f} = \frac{1}{N_b} \sum_{i=1,N_b} \hat{f}_i$$

et l'erreur Jack knife $\sigma_{JK}$ est donnée par:

$$\sigma_{JK}^2 = \frac{N_b - 1}{N_b} \sum_{i=1,N_b} \left( \hat{f}_i - \bar{f} \right)^2$$

La véritable erreur est obtenue en augmentant la longueur $b$ du bloc jusqu'à atteindre un plateau. La valeur plateau satisfait (approximativement)

$$\frac{\sigma_{JK}^2}{\sigma^2} = 2\tau_{int}$$

et donc on obtient une autre estimation du temps d'autocorrélation.

L'analyse du Jack knife est aussi valable pour les quantités secondaires, c'est à dire qu'elle prend en compte la propagation des erreurs. C'est la raison pour laquelle elle est très utilisée, même lorsqu'il n'y a pas de problèmes de l'autocorrélation. Si une observable $G$ est une fonction de $f, g, ...$ alors la valeur moyenne est estimée selon:

$$\bar{G} = \frac{1}{N_b} \sum_{i=1,N_b} G(\hat{f}_i, \hat{g}_i, ...)$$

et l'erreur est:

$$\sigma_{JK}^2(G) = \frac{N_b - 1}{N_b} \sum_{i=1,N_b} \left( G(\hat{f}_i, \hat{g}_i, ...) - \bar{G} \right)^2$$

### 1.7.2 Exercice 5

Complèter le programme de l'exercice précédant pour calculer l'erreur Jack knife en fonction de la taille de bloc. Comparer la valeur du plateau à $\sigma\sqrt{2\tau_{int}}$ où $\tau_{int}$ est la valeur déterminée par l'étude de la fonction d'autocorrélation. Modifier le code pour que l'algorithm de Metropolis effectue $n$ itérations à vide avant d'accepter la nouvelle valeur de $x$ et observer l'évolution de l'erreur Jackknife quand $n$ augmente (voir l'exemple de la Figure 1.5).

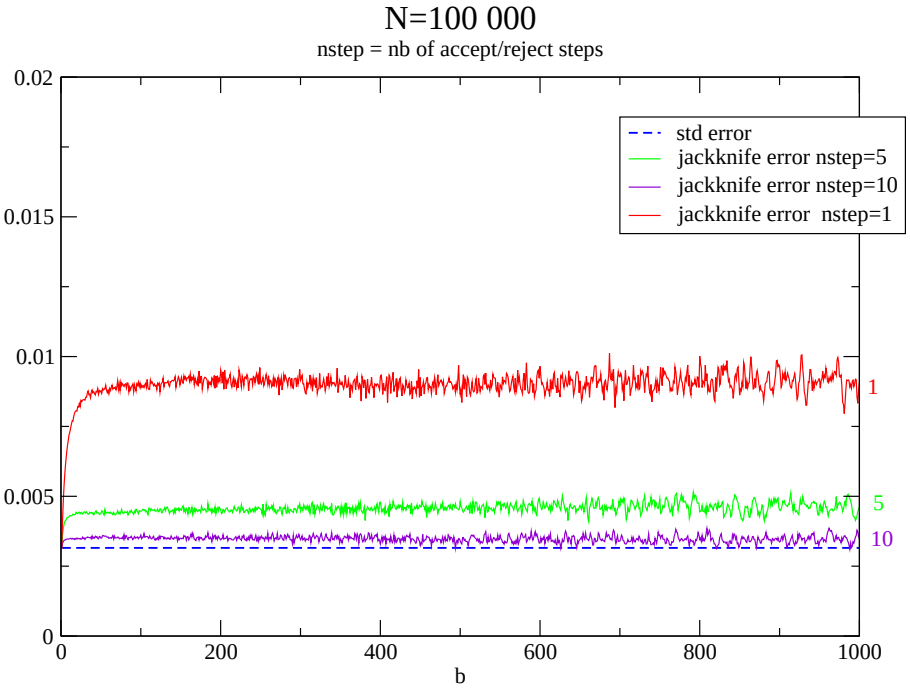
N=100 000
nstep = nb of accept/reject steps
std error
jackknife error nstep=5
jackknife error nstep=10
jackknife error nstep=1
0.02
0.015
0.01
0.005
0
0
200
400
600
800
1000
b
1
5
10

Figure 1.5:

# Bibliography

[1] B.A. Berg, Markov Chain Monte Carlo Simulations and Their Statistical Analysis, World Scientific, Singapore, 2004.
On trouvera une introduction à l'URL: www.worldscibooks.com/mathematics/etextbook/5904/5904_intro.pdf

# Chapter 2

# TP2: QCD sur réseau

## 2.1 Prélude

Dans ce deuxième TP on va appliquer les méthodes étudiées précédemment à des calculs semi-réalistes de QCD sur réseaux. Le terme semi-réaliste est du au choix délibéré de faire ces calculs avec des machines de type ordinateurs personnels. Ces machines sont suffisamment rapides pour obtenir des résultats intéressants dans un temps raisonnable mais à condition de limiter ses ambitions:

1. La principale limitation est qu'on impose aux quarks d'être *statiques*. Cela veut dire qu'ils interagissent avec le champ de gluons, ce qui permet de calculer la force entre quarks, mais ils ne bougent pas sous l'influence de cette force.

2. Une autre limitation est la taille du réseau, c'est à dire le nombre de sites dans chaque dimension. Si on veut avoir un temps de calcul raisonnable on ne peut guère dépasser $14^4$. En même temps, pour que la théorie sur réseau soit une bonne approximation de la théorie continue, il faut que la maille du réseau soit suffisamment petite. Si on prend $a = 0.1fm$ pour fixer les idées, alors la dimension physique du réseau est au mieux $14*0.1 = 1.4fm$ , trop petit pour y mettre un vrai hadron dont le rayon est environ $0.8fm$. Il faut donc se limiter à calculer des quantités qui tiennent dans un petit volume. C'est le cas de la boucle de Wilson, qui permet de calculer la force entre deux quarks statiques. Comme c'est un objet à 2 dimensions, on peut dimimuer les dimensions perpendiculaires à la boucle et augmenter les autres tout en gardant un nombre de sites raisonnable. Il ne faut pas imaginer pour autant qu'on a complètement éliminé le problème car, même si la boucle tient dans le réseau, le simple fait que celui ci a un volume fini perturbe la distribution du champ de gluons et ceci introduit des erreurs systématiques dans le calcul de l'énergie. Il faut être conscient de ces effets, dit "de volume fini", quand on essaie de confronter les quantités calculées à l'expérience. Par contre ils ne changent pas qualitativement les résultats,

comme par exemple le confinement des quarks et leur libération à haute température.

3. Pour accélérer les calculs on supposera la plus part du temps que le nombre de couleurs est égal a $N_c = 2$ au lieu de 3 car les phénomènes physiques ne changent pas qualitativement quand on passe de l'un à l'autre. De toute façon la bibliothèque FermiQCD permet de calculer avec $N_c > 1$ quelconque en cas de besoin. C'est juste une question de temps.

4. Le calculs sont faits par la méthode de Monte-Carlo dont l'erreur décroit comme $1/\sqrt{N}$ où $N$ est le nombre de tirages. Passer de 1 heure à 4 heures de calcul peut être prohibitif, alors que cela n'apporte qu'un facteur 2 sur l'erreur. Il faut donc être raisonnablement exigeant sur la précision statistique et ne pas oublier qu'il y a d'autres sources d'erreurs, comme le volume fini.

On va donc étudier la boucle de Wilson et en déduire le potentiel d'interaction entre un quark et un anti-quark. Comme le temps est limité et qu'on ne peut pas le gaspiller à traquer les erreurs de code, cette séance de TP consistera surtout à faire tourner des codes qui ont été mis au point pour la circonstance, et dont les fonctions sont résumées dans les Sections 2.5,2.6,2.7. Il y aura donc un coté presse-bouton, mais ce sera aussi l'occasion de voir comment se développe un vrai calcul, comment on choisit les paramètres, comment on extrait la physique, etc... De plus ces codes tout prêts serviront aussi d'exemples d'utilisation des fonctions disponibles dans FermiQCD pour ceux qui veulent développer un projet personnel.

## 2.2 Rappel sur la boucle de Wilson et le potentiel quark anti-quark

Si on note $U(x,\mu)$ la matrice de $SU(N_c)$ associée au lien qui part du site $x = (x_0, x_1, x_2, x_3)$ et va dans la direction ($\mu = 0, 1, 2$ ou 3), la boucle de Wilson pour un contour $C_x = \{x \to x+\mu_1 \to x+\mu_1+\mu_2 \ldots \to x-\mu_L \to x\}$ est définie par le produit matriciel

$$W(C_x;U) = U(x,\mu_1)U(x+\mu_1,\mu_2)U(x+\mu_1+\mu_2,\mu_3)\ldots U(x-\mu_L,\mu_L) \quad (2.1)$$

c'est à dire le produit des matrices des liens qui définissent le contour. Sa valeur moyenne dans l'état fondamental (ou vide) de QCD est

$$< W(C_x) >= \frac{1}{Z}\int [dU] e^{-S(U)} W(C_x, U) \quad (2.2)$$

Comme le vide est invariant par translation, la valeur moyenne $< W(C_x) >$ne dépend pas du point d'origine $x$ et on peut remplacer $W(C_x, U)$ par la moyenne

sur toutes les positions $x$:

$$< W(C) > = \frac{1}{Z}\int [dU] e^{-S(U)} W(C,U) \tag{2.3}$$

$$W(C,U) = \frac{1}{N_x}\sum_x W(C_x,U) \tag{2.4}$$

En pratique le réseau n'est pas infini et pour minimiser les effets de surface on impose aux matrices de lien de satisfaire des conditions périodiques. Par exemple si la longueur dans la direction 0 est $L$ on aura

$$U([(x_0,x_1,x_2,x_3),\mu)] = U[(x_0+L,x_1,x_2,x_3),\mu] \tag{2.5}$$

ce qui revient à faire une copie du réseau dans la direction 0. Sur la Figure 2.1 on montre comment une boucle qui déborderait du réseau est en fait calculée à partir des valeurs des liens intérieurs décalés de $L$. Tout se passe comme si on avait enroulé la feuille et collé les 2 bords pour faire le cylindre montré sur la Figure 2.1. Si on enroule aussi les autres dimensions on obtient un tore à 4 dimensions, comme on le montre sur la Figure 2.1, pour le cas à 2 dimensions. On voit bien dans cette représentation que la position de la boucle sur le tore n'a pas d'importance dans la mesure où ses dimensions sont petites devant les rayons de courbure de la surface du tore.

On rappelle que le potentiel d'interaction $V(r)$ entre un quark et un anti-quark statiques est relié à la valeur moyenne d'une boucle rectangulaire de cotés $(t,r)$ suivant

$$< W(t,r) > \rightarrow C(r)e^{-V(r)t} \text{ qd } t\rightarrow\infty \tag{2.6}$$

Sur le réseau les distances varient de façon discrète puisque la maille $a$ est finie. On a $r = n_r a,\ t = n_t a$ avec $n_r$ et $n_t$ entiers et la relation s'écrit

$$< W(n_t,n_r) > \rightarrow C(an_r)e^{-aV(an_r)n_t} \text{ qd } n_t\rightarrow\infty \tag{2.7}$$

ou encore

$$aV(an_r) = \lim_{n_t\rightarrow\infty} Log\frac{< W(n_t-1,n_r) >}{< W(n_t,n_r) >} \tag{2.8}$$

Pour pouvoir déterminer la limite $n_t\rightarrow\infty$ il faut que la dimension temporelle du réseau soit suffisamment grande.[1] On doit donc vérifier que $Log[< W(n_t-1,n_r) > / < W(n_t,n_r) >]$ atteint un plateau, aux erreurs statitstiques près. Si ce n'est pas le cas c'est que la valeur maximale de $n_t$ n'est pas assez grande pour la distance $n_r$choisie. Si, à cause du temps de calcul, on ne peut pas augmenter $n_t$, alors le potentiel ne peut pas etre calculé pour cette distance. Si on observe un plateau on fait un fit de la forme $\gamma + \delta\exp(-\omega n_t)$, et la limite cherchée est égale à $\gamma$.

[1] C'est aussi une condition nécessaire pour pouvoir considerer que le réseau est à température nulle.

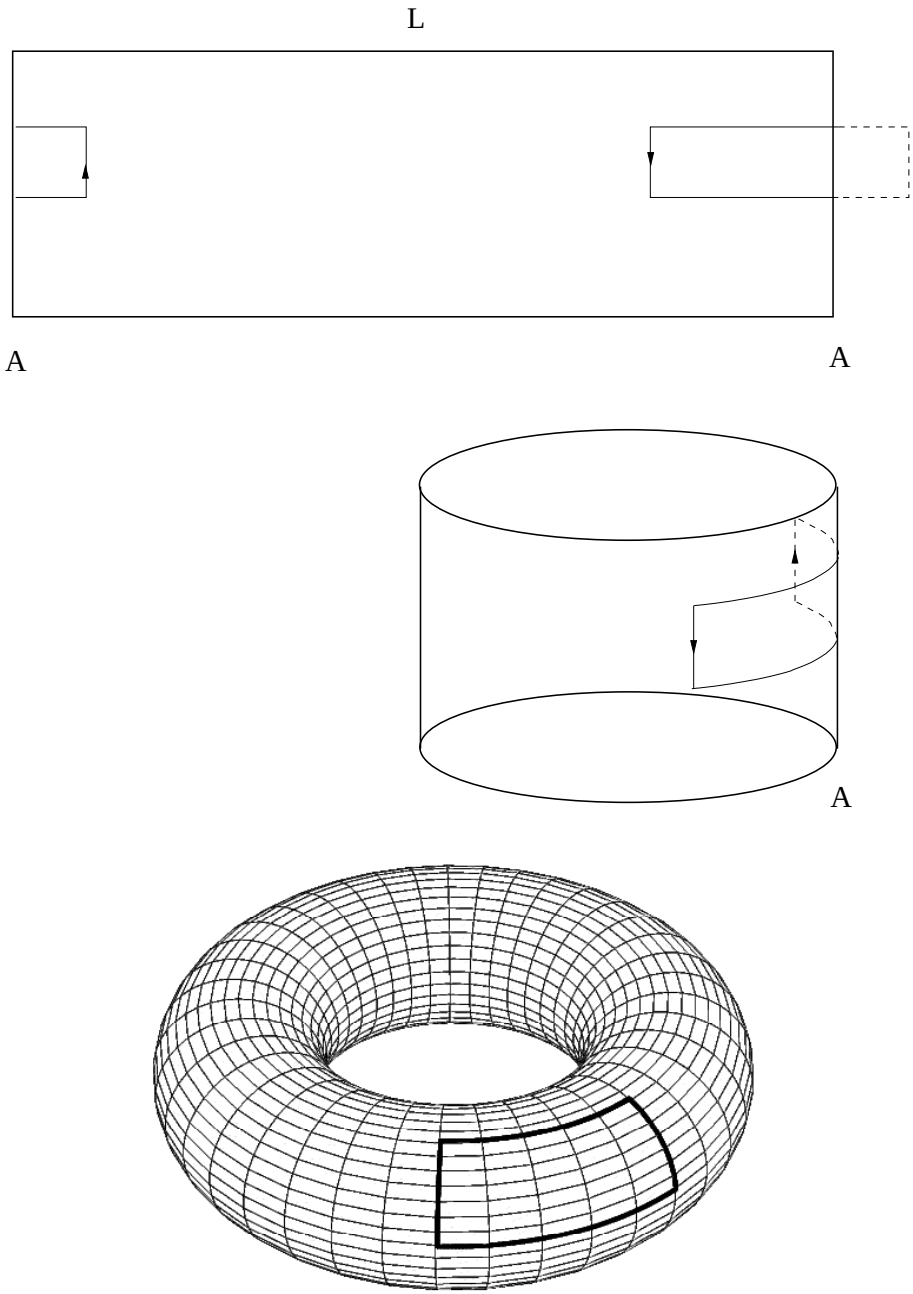


Figure 2.1:

## 2.3 La maille du réseau

Le réseau permet donc de déterminer la quantité sans dimension $aV(an_r)$. Remarquons que cette quantité ne peut pas dépendre explicitement de $a$. Pour s'en convaincre il suffit de considérer l'expression 2.2 qui sert à calculer $< W(n_t, n_r) >$. La boucle dans l'intégrale ne dépend que de $(n_t, n_r)$ et l'action $S(U)$, qui est une quantité sans dimension, ne peut pas dépendre de $a$ puisque dans QCD il n'y a pas de paramètre dimensionnel pour compenser. Le seul paramètre dont dépend $aV(an_R)$ est donc $g$, la constante de couplage de QCD qui apparait dans l'action $S(U)$. Comme on utilise plutôt le paramètre $\beta = 2N_c/g^2$ on écrira:

$$aV(an_r) = v(n_r, \beta).$$

Le potentiel physique s'écrit

$$V(r) = \frac{1}{a} v\left(\frac{r}{a}, \beta\right) \tag{2.9}$$

et on voit que pour le calculer il faut connaitre la valeur de $a$. Pour la déterminer on suppose que le potentiel est de la forme

$$V(r) = c_0 + \sigma r + -\frac{e}{r} \tag{2.10}$$

comme le suggère la phénoménologie des mésons lourds. On verra que c'est bien la forme que produit le calcul sur réseau. Les constantes $\sigma$ et $e$ valent approximativement:

$$\sigma \simeq (450MeV)^2, \ e \simeq 0.45$$

Le terme constant $c_0$ n'a pas de signification puisqu'une énergie potentielle est déterminée à une constante additive près. Une autre façon de le dire est que c'est seulement la force $\vec{\nabla} V$ qui est mesurable. La constante $\sigma$ est appellée tension de corde est c'est le terme $\sigma r$ qui produit le confinement. En unités macroscopiques on a $\sigma \sim 10\,\text{tonnes}$. Une fois que l'on a obtenu $v(n_r, \beta)$ pour une valeur donnée de $\beta$ on écrit:

$$V(r) = \frac{1}{a} v\left(\frac{r}{a}, \beta\right) = c_0 + \sigma r - e/r \tag{2.11}$$

ou encore

$$v(n_r, \beta) = ac_0 + \sigma a^2 n_r - e/n_r \tag{2.12}$$

et il suffit de faire un fit de $v(n_r, \beta)$ de la forme:

$$v(n_r, \beta) = A_0(\beta) + A_1(\beta) n_r - A_{-1}(\beta)/n_r \tag{2.13}$$

pour en déduire

$$a = \sqrt{\frac{A_1(\beta)}{\sigma}}, \ e = A_{-1}(\beta). \tag{2.14}$$

Comme le réseau n'est qu'un intermédiaire de calcul on doit en principe prendre la limite $a \to 0$ pour retrouver un espace-temps continu mais ce n'est pas

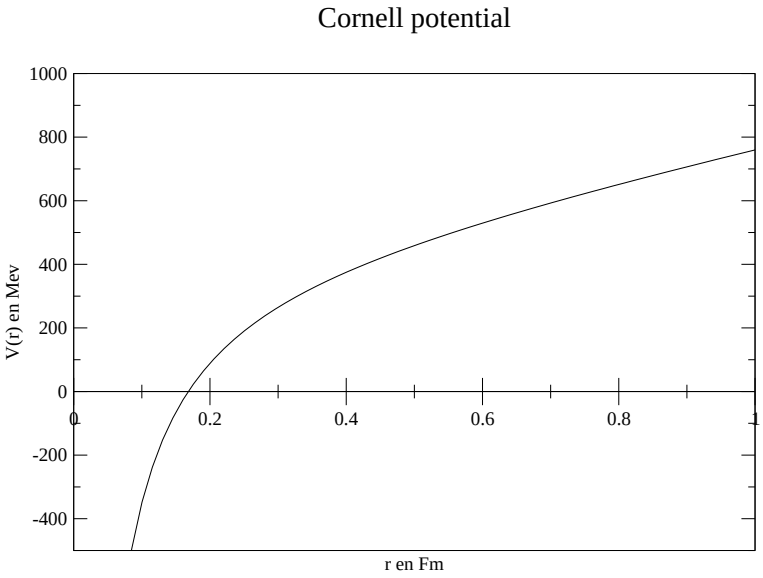


Figure 2.2:

possible en pratique car, si on veut conserver la même dimension physique au réseau, il faut que le nombre de points augmente comme $1/a$. Il faut donc être pragmatique et choisir une valeur de $a$ raisonnable, suffisamment grande pour que la physique tienne dans le réseau et suffisamment petite pour que les erreurs dues à la discrétisation soient supportables. Pour le problème qui nous intéresse on est guidé par l'allure du potentiel phénoménologique (2.10) qui est tracé sur la Figure 2.2. La région dominée par le confinement commence vers $r = 0.4fm$. Donc il faudrait calculer le potentiel jusqu'à au moins $0.6fm$ pour avoir un résultat convaincant. Ceci implique un réseau d'au moins $1fm$ pour limiter les effets de taille finie. D'autre part si on veut échantillonner la région en $1/r$ avec une précision acceptable, la maille doit être de l'ordre de 0.1fm.

Comme on le voit sur l'eq. (2.14) c'est $\beta$ qui détermine la valeur de $a$. Celà veut dire que $a$ ne sera connu *qu'à la fin du calcul.* Si la valeur n'est pas bonne on aura calculé pour rien. Il faut donc procèder par essais, en essayant de dégrossir la relation entre $\beta$ et $a$ avec des calculs pas trop monstrueux en temps CPU.

## 2.4 Utilisation de FermiQCD

#### 2.4.0.1 Introduction

On évalue la quantité

$$< W(C) >= \frac{1}{Z} \int [dU] e^{-S(U)} W(C,U)$$

en créant une chaine de Markov $\{U^{(1)}, U^{(2)}, ...U^{(N)}\}$, où $U^{(i)}$ représente l'ensemble des matrices de lien à chaque étape de la chaine, et telle que

$$\frac{1}{N}\sum_i W(C, U^{(i)}) \rightarrow < W(C) > \quad \text{quand } N \rightarrow \infty$$

Dans le jargon des réseaux l'ensemble des liens $U^{(i)}$ est une "configuration de jauge". Nous allons utiliser le générateur fourni par FermiQCD, qui est basé sur la méthode du bain thermique, sans entrer dans le détail de son fonctionnement. On trouvera un exposé detaillé de la mise en oeuvre de l'algorithm dans la référence [1]. Pour nous c'est une boite noire à laquelle on soumet une configuration de jauge $U^{(i)}$ et qui nous retourne une autre configuration $U^{(i+1)}$ générée à partir de $U^{(i)}$ par le bain thermique. Le calcul est amorcé en proposant une configuration initiale $U^{(0)}$ qui est assez arbitraire. Par exemple:

- Démarrage à froid: $U^{(0)} = 1$ pour tous les liens.
- Démarrage à chaud: chaque lien de $U^{(0)}$ est une matrice tirée au hazard.
- Démarrage en pseudo équilibre: $U^{(0)}$ est la dernière configuration d'un autre calcul fait avec une valeur de $\beta$ légèrement différente (utile pour étudier la dépendance en température)

#### 2.4.0.2 Instructions essentiellea

Il n'est pas question de faire une revue de FermiQCD. Pour apprendre, ou pour trouver un renseignement, le plus simple est de regarder les exemples de ce TP ou ceux qui accompagnent la librairie. On trouvera aussi des renseignements à l'URL *http://www.fermiqcd.net/*. Ce qui suit explique seulement les instructions les plus utilisées dans la suite.

```
   //
   int box[]={16,6,6,12};        //defines the lattice dimensions (voir
la remarque plus bas)
   mdp_lattice lattice(4,box);   //creates a lattice in 4 dimensions
                                 //named "lattice"
   //
   gauge_field U(lattice,2);    //creates the gauge field U for 2 colors
   coefficients gauge;          //creates an object "coefficients"

                                //which serves to store the couplings
   gauge["beta"]=2.7;           //set the coupling to beta=2.7
   //
   set_hot(U);                 //creates a hot configuration
   //
   WilsonGaugeAction::heatbath(U,gauge,1);    //takes the current U
                                             //as input and returns
```

```
                                          //a new one after
                                          //1 heatbathstep
                                          //for each link
WilsonGaugeAction::heatbath(U,gauge,5);    //takes the current U
                                          //as input and returns

                                          //a new one after
                                          //5 heatbathstep
                                          //for each link
```

Le dernier argument dans l'appel ci-dessus (paramètre de décorrélation) force l'exécution du bain thermique un certain nombre de fois avant de retourner la nouvelle configuration.

Quelques instructions souvent utilisées:

```
//
site x(U.lattice());       // creates x as a lattice site
x.set(1,2,3,6);            // set x to (1,2,3,6)
cout<<U(x,3)<<endl;        //print the value of U
                           //for the link starting at x=(1,2,3,6)

                           //and going in the direction 3
x=x+2;                     //displace x by 1 unit
                           //in the direction 2.
                           //Here (1,2,3,6) ->(1,2,4,6)
forallsites(x){...}        //creates a loop over all sites.
//
```

#### 2.4.0.3 Quelques fonctions associées aux objets

L'avantage d'un langage orienté objet est qu'un objet transporte avec lui les paramètres dont il dépend. Voilà quelques exemples:

- `U.nc` est le nombre de couleurs.
- `U.lattice()` est le réseau sur lequel U existe.
- `U.lattice().nx[1]` donne la dimension du réseau dans la direction 1.
- `U.lattice().nvol_gl` est le volume du réseau, c'est a dire le nombre de sites.
- `U.save("myfield")` écrit la configuration $U$ dans le fichier myfield.
- `U.load("myfield")` lit la configuration $U$ dans le fichier myfield.

#### 2.4.0.4 Utilisation de FermiQCD pour le calcul de l'erreur

Supposons qu'on veuille calculer la valeur et l'erreur de:

$$\frac{1}{dt} Log \frac{< W(n_t - dt, n_r) >}{< W(n_t, n_r) >} \tag{2.15}$$

avec

$$< W(n_t, n_r) >= \frac{1}{N} \sum_{i=1,N} W(n_t, n_r, U^{(i)})$$

En géneral $dt = 1$ mais il peut être utile de généraliser. On crée d'abord une fonction[2]

```
// function for jack knife
float f1(float *x, void *a)
  {float dt=*((float*)a);
   return log(x[0]/x[1])/dt;
  }
//
```

Ensuite on crée un conteneur pour stocker les valeurs de $W(n_t, n_r, U^{(i)})$:

```
mdp_jackboot jack(N+1,2);
```

Le conteneur `jack` va contenir les $N$ valeurs[3] $W(n_t - dt, n_r, U^{(i)})$ et $W(n_t, n_r, U^{(i)})$. Par exemple

```
for(i=1;i<=N;i++)
{jack(i,0)=W(n_t - dt, n_r, U^(i));
jack(i,1)=W(n_t, n_r, U^(i));
}
```

On indique ensuite que c'est la fonction `f1` qui doit être utilisée pour combiner les 2 boucles:

```
jack.f=f1;
```

et on a :

$$\frac{1}{dt} Log \frac{< W(n_t - dt, n_r) >}{< W(n_t, n_r) >} = \text{jack.mean();}$$
$$\text{Jack knife error} = \text{jack.j\_err();}$$

On pourra trouver cette séquence d'instruction dans les codes fournis. Notons qu'à cause des fluctuations statistiques l'argument du Log peut être négatif au début de l'itération, et donc le code retourne `nan` (not a number). Il faut évidemment éliminer ces points pathologiques avant de faire le fit pour extraire le potentiel.

[2] La valeur de dt est transmise par le void pointeur a, d'où l'instruction un peu compliquée pour la récupérer (casting+dereferencing).

[3] l'indice 0 n'est pas utilisé mais il faut réserver la place

#### 2.4.0.5 Utilisation de FermiQCD pour les fits

Supposons que pour $n$ points $\{x_0, x_1, ...\}$ on a $n$ valeurs $\{y_0, y_1, ...\}$ ainsi que les erreurs $\{dy_0, dy_1, ...\}$. On veut ajuster une fonction $f(x,\ p_0, p_1, ...)$ à $q$ paramètres en minimisant la quantité:

$$\sum_{i=0,n-1} \left( \frac{y_i - f(x_i, p_0, p_1, ...)}{dy_i} \right)^2$$

par rapport aux paramètres.

On definit d'abord la fonction à ajuster. Ici on prend comme exemple celle qui sert à ajuster la dépendence en temps de $v$:

```
float fitlog(float x, float *p, long ma, void *junk)
  {return p[0]+p[1]*exp(-p[2]*x);}
```

On utilise un objet de la classe `Measure` pour stoquer $\{y_i\}$ et $\{dy_i\}$. Par contre les abcisses sont dans un simple tableau:

```
Measure ydy[n];
float x[n];
for(i=0;i<n;n++)
  {x[i]=x_i                    //abcisses
   ydy.set[i].set(y_i,dy_i);   //ordonnées et erreurs
  }
```

On crée un tableau pour les paramètres et pour la matrice de covariance (c'est un objet, pas un tableau!)

```
float p[q]
mdp_matrix covar(q,q);
```

On initialise les paramètres, par exemple

```
p[0]=1;p[1]=1;...
```

et on appelle la fonction

```
BaesyanLevenbergMarquardt(x,ydy, 0,n-1, p, q, covar, fitlog);
//
```

On peut remplacer les arguments $(0, n-1)$ par $(a > 0, b < n-1)$ auquel cas le fit sera fait sur les données qui vont de $a$ à $b$.

Le résultat est $\{p[0] \pm \sqrt{covar[0,0]}, p[1] \pm \sqrt{covar[1,1]}, ...\}$.

Notons que cette facon d'ajuster la fonction n'est pas la plus générale car elle suppose qu'il n'y a pas de corrélations entre les données. On s'en contentera.

#### 2.4.0.6 Calcul des boucles

FermiQCD fournit des fonctions pour calculer des boucles de forme quelconque mais elles ne sont pas très commodes a utiliser. On utilisera plutot celles qui calculent seulement des boucles rectangulaires. Pour une boucle située dans le plan (mu,nu) de longueur (lmu,lnu) la boucle de Wilson pour la configuration $U$ est donnée par la fonction

```
my_average_loop(U,mu,lmu,nu,lnu)
```

**Changement dans la version 2009: dans la partie Confinement, si** systemchoice==2 **le programme calcule effectivement avec** `my_average_loop`. `Si` systemchoice==0 **alors le boucles sont calculées en une seule passe par** make_aver_loops_with_CPU **qui est beaucoup plus rapide.**

On dispose aussi de la boucle de Polyakov (voir section 2.8.2) qui est donnée par

```
my_polyakov_loop(U)
```

Il existe aussi des fonctions pour calculer une boucle sans moyenner sur la position mais, en principe on n'en a pas besoin.

#### 2.4.0.7 Le parallèlisme

FermiQCD est conçue pour fonctionner en mode parallèle. Le programme est parsemé d'instructions qui assurent la communication entre les processeurs. Comme on travaille sur des machines mono processeur ces instructions ne servent à rien mais il faut les inclure pour que le code puisse fonctionner. Dans les exemples elles sont repérées par un commentaire explicite.

## 2.5 La thermalisation de la chaine de Markov

Le programme proposé effectue le calcul de 2 boucles typiques sur un reseau $8^4$ pour $SU(2)$, $\beta = 2.3$ et visualise le résultat en ligne en fonction du nombre d'itérations. Le bain thermique est appelé une seule fois par itération. On peut observer comment la valeur calculée approche la valeur limite et déterminer le nombre d'itération qu'il faut faire à vide avant de commencer à calculer. A la fin du calcul le code montre à nouveau le résultat avec des barres d'erreurs *standard*.

Le programme est Thermalise.cpp dans le directory Thermalisation. La commande `make` fait la compilation et l'assemblage. L'execution est lancée par `./a.out` . Pour modifier un paramètre il faut éditer le fichier Thermalise.cpp et le recompiler. Pour faire des modifications plus importantes du code il est recommandé de sauver Thermalise.cpp sous un autre nom et d'adapter le fichier MakeFile en conséquence (NB: la commande `make` sans argument exécute l'instruction `all` du MakeFile, tandis que la commande `make machin` execute `machin`)

## 2.6 Autocorrelations et erreur Jackknife

Le résultat d'un calcul sur réseau est sans intérêt si on ne peut pas lui affecter une erreur crédible. Nous allons mettre en pratique les notions d'autocorrelation et d'erreur Jack knife étudiées précédemment. Le code Autocorrel.cpp évalue la boucle de Wilson $4 \times 4$ sur un réseau $8^4$ pour $SU(2)$, $\beta = 2.3$. Idéalement il faudrait faire cette étude de façon un peu plus systématique, c'est à dire en considérant plusieurs valeurs de $\beta$, plusieurs tailles de réseau et de boucles mais pour l'instant on se contentera de cet exemple. L'étude est a refaire lorsqu'on a définitivement choisi les paramètres du calcul, ce qui peut conduire à changer le paramètre de décorrélation (voir 2.4.0.2).

Le premier graphe montre simplement l'évolution de la valeur moyenne en fonction du nombre d'itérations. Le second, qui apparait lorsque les 500 itérations sont terminées, montre l'erreur Jack knife en fonction de la taille du bloc enlevé (voir TP1). Le troisième est la fonction d'autocorrélation pour cette boucle (on se souvient que cette fonction est définie pour chaque observable) et le quatrième montre le temps d'autocorrélation integré.

Il est plus commode de calculer systématiquement l'erreur Jack knife avec une valeur de bloc égale à 1. C'est ce qui est fait dans FermiQCD. En conséquence, pour avoir une estimation correcte de l'erreur, il est indispensable d'ajuster le paramètre de décorrélation selon le résultat trouvé pour la fonction d'autocorrélation. Dans le cas présent on voit que ce n'est pas nécessaire: la fonction d'autocorrélation tombe à zéro dès le premier point, ce qui est confirmé par la quasi saturation de l'erreur Jack knife pour un bloc de taille unité.

*Par contre la situation peut être très différente quand on étudie la transition de déconfinement avec la boucle de Polyakov. De facon générale les transtions de phase posent des problèmes de convergence à cause des grandes fluctuations du paramètre d'ordre au voisinage de la transition. Ces fluctuations de longue portée tendent à corréler des parties du réseau pourtant distantes. Il faut alors soigneusement étudier l'autocorrélation.*

## 2.7 Le calcul du potentiel

(Rappel: dans Confine.cpp la taille du réseau et des boucles est définie dans parameters.h à partir de 2009)

Une fois qu'on a à peu près déterminé le temps de thermalisation et de décorrélation, on peut entrer dans le vif du sujet et calculer le potentiel entre un quark et un anti-quark. C'est ce que fait le programme Confine.cpp

On lui spécifie un certain nombre de points dans la direction 3, par exemple:

```
int zvalue[]={1,2,3,5,7,9,11};
```

qui seront les distances entre le quark et l'anti-quark. On fait de même pour la direction 0 (le temps)

```
int tvalue[]={1,2,3,5,7,9,11};
```

**Changement dans la version 2009: pour des raisons d'optimisation du calcul les tableaux** `zvalue[] et tvalue[] sont maintenant fixes et valent {1,2,3,4,5,6,7,8,9,10}`.

Après avoir effectué la thermalisation:

```
for(int k=1;k<=nheat;k++){WilsonGaugeAction::heatbath(U,gauge,1);}
```

le programme effectue une boucle d'itérations où, pour chaque nouvelle configuration de jauge, il calcule les boucles de Wilson dans le plan (0,3) de taille `(tvalue,zvalue)` et les stocke.

Toutes les `kupdate` itérations, il calcule la quantité:

$$\frac{1}{dt} Log \frac{< W(n_t - dt, n_r) >}{< W(n_t, n_r) >}$$

pour toutes les valeurs de $n_t = tvalue$, $n_r = zvalue$, avec $dt$ égal à l'intervalle entre $n_t$ et la précédente valeur. Pour chaque `zvalue` il ajuste la fonction $\gamma + \delta \exp(-\omega n_t)$ et en déduit le potentiel et son erreur. Lorque le potentiel a été calculé pour tous les `zvalue` le programme lui ajuste la fonction $ac_0 + \sigma a^2 n_r - e/n_r$ et en déduit la valeur de la maille $a$. Les résultats sont affichés en ligne par Grace. Le graphes $0 \rightarrow 5$ correspondent chacun à une valeur de `zvalue` et montrent la dépendance en temps ainsi que le fit. Le graphe 6 montre le temps CPU consacré au bain thermique et au calcul des boucles. Le graphe 7 montre la valeur de $a$ en fonction du nombre d'itération, ce qui donne une bonne idée de la convergence. Le graphe 8 montre le potentiel et le fit.

Les paramètres et les résultats du calcul sont écrits dans un fichier dont le nom est incrémenté pour éviter les écrasements accidentels. Pour retrouver le fichier de résultats correspondant à telles valeurs des paramètres il suffit d'utiliser la fonction `grep.` Exemple:

```
grep -d read 'lattice' ./Res*
```

affiche toutes les lignes contenant la chaine "`lattice`" dans n'importe quel fichier de type `Res` dans ce répertoire.

Les paramètres du calcul sont bien mis en évidence dans le code. Les valeurs par défaut permettent de faire un premier calcul dans un temps raisonnable (1/2 heure). Lorsque le nombre d'itération maximum est atteint, le programme propose d'éventuellement d'en rajouter.

## 2.8 Projets

### 2.8.1 Projet 1: Etude du confinement

Ce projet ne demande aucun développement de code. Il est démandé d'utiliser Confine.cpp pour déterminer la relation entre la maille du réseau et la constante

de couplage nue $\beta$. Il faudra d'abord choisir un réseau de taille modeste pour dégrossir la region intéressante, c'est à dire $a = 0.05fm$ à $a = 0.3fm$ et ensuite faire une étude plus fine. Contrôler les effets de taille finie pour une ou deux valeurs de $\beta$.

1. Tracer la courbe et comparer à la prédiction perturbative:

$$a = C^{ste} \exp(-\frac{6\pi^2}{11N_c}\beta).$$

   (On rappelle que la théorie perturbative prédit comment *varie* $a$ en fonction de $\beta$ mais ne donne pas la constante.)

2. Choisir $\beta$ correspondant à $a = 0.1fm$ et tracer le potentiel en unités physiques.

3. Comparer le coefficient du terme coulombien (en $1/r$) à la valeur expérimentale.

### 2.8.2 Projet 2: Etude du déconfinement

Dans ce projet on veut étudier la transition de déconfinement en fonction de la température. Il faudra d'une part adapter le code d'étude de l'autocorrélation (changement mineur) et surtout faire un nouveau code (en fait bien plus simple que Confine.cpp) pour étudier l'évolution de la boucle de Polyakov en fonction de $\beta$, qui joue le rôle de temperature.

On rappelle (voir le cours) que la dimension temporelle du réseau est l'inverse de la température et que la boucle de Polyakov est définie par:

$$< P(\vec{x}) >= \frac{1}{Z} \int [dU] e^{-S(U)} U[(0, \vec{x}), 0] U[(a, \vec{x}), 0] U[(2a, \vec{x}) ... U[(aN_0, \vec{x}), 0]$$

c'est à dire le produit des matrices de lien pour une ligne ( positionnée en $\vec{x}$ ) qui va d'un bout à l'autre du réseau dans la direction 0 (le temps). En raison de la périodicité, on peut voir cette ligne comme une boucle fermée qui entoure le tore. En invoquant l'invariance du vide par translation dans l'espace à 3 dimensions on peut remplacer la boucle au point $\vec{x}$ par sa moyenne:

$$< P >= \frac{1}{N_1 N_2 N_3} \sum_{\vec{x}} < P(\vec{x}) >$$

où $N_1 N_2 N_3$ est le volume spatial.

On peut montrer que $P$ est un paramètre d'ordre dans le sens que $P = 0$ dans la phase confinée. Lorsque la température $T = 1/(N_0 a)$ augmente on s'attend à ce que $P$ devienne non nul au dessus d'une température critique $T_c$. La facon naturelle d'augmenter la température serait de diminuer $N_0$ mais pour $a$ fini, une variation de 1 de $N_0$ correspond à une variation discrète de la température, ce qui n'est pas commode. On préfère fixer $N_0$ et faire varier

$a$, c'est à dire $\beta$. On sait, et on l'a verifié explicitement au cours de ce TP, que $a$ diminue quand $\beta$ augmente. Donc une augmentation de $\beta$ correspond effectivement à une augmentation de température. Pour que l'interprétation thermodynamique soit correcte il faut que le volume spatial reste grand quand $a$ diminue. Le critère est que la taille spatiale ($aN_i$, $i-1,2,3$) du réseau doit être beaucoup plus grande que l'inverse de la tempèrature $N_0 a$. Donc il faut choisir $N_0 \ll N_i$, $i = 1,2,3$. Typiquement on pourra prendre $N_0 = 4$, $N_i = 8 \div 10$.

Le but du projet est de tracer le module (pourquoi?) de $< P >$ en fonction de $\beta$ pour $SU(2)$ ou/et $SU(3)$, d'observer la transition de phase et de déterminer la température critique en MeV (On pourra éventuellement aussi regarder ce qui se passe dans un réseau à 2 dimensions car le calcul est très rapide). On étudiera d'abord l'autocorrélation et on choisira le paramètre de décorrélation en conséquence. A chaque incrément de la température on prendra comme configuration de jauge initiale la dernière de la température précédente.

## 2.9 Contrôle du travail

1. Pour le TP1: rédiger vos résultats pour l'exercice 5.
2. Pour le TP2: rédiger vos observations pour les sections 2.7,2.15,2.9.
3. Mettre en oeuvre un des deux projets proposés (section 2.10) et rédiger vos résultats.

Les points 1,2 constituent la moitié de la note de TP.

# Bibliography

[1] Montway, I. and Munster, G., Quantum Fields on a Lattice. , Cambridge University Press (1994)